软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2016,27(9):2426−2442 [doi: 10.13328/j.cnki.jos.004859]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种面向实时系统的程序基本块指令预取技术*

王恩东, 倪 璠, 陈继承, 王洪伟, 唐士斌

(高效能服务器和存储技术国家重点实验室(浪潮(北京)电子信息产业有限公司),北京 100085)

通讯作者: 倪璠, E-mail: nifan@inspur.com

摘 要: 面向通用计算机系统的指令预取技术无法满足实时系统的应用需求,其中一个重要原因是:无效预取引起的指令 Cache 内容污染使得实时任务 WCET 评估值不够精确,导致系统可调度性下降,严重影响系统效率.以简化实时任务 WCET 分析、降低任务 WCET 评估值为目标,提出一种基于程序基本块的指令预取方法.该方法以基本块为粒度执行指令预取,避免了传统指令预取技术引入的无效预取;通过简化最坏情况下的指令访问命中/缺失情况判定,简化任务 WCET 分析过程并优化 WCET 评估值.实时基准测试程序评估结果表明:与常规无预取方法相比,该预取方法可使实时任务 WCET 评估值降低约 20%,平均执行情况下的指令 Cache 访问性能提升约 10%.

关键词: 实时系统;最坏情况执行时间;高速缓存性能;指令预取;基本块

中图法分类号: TP316

中文引用格式: 王恩东,倪璠,陈继承,王洪伟,唐士斌.一种面向实时系统的程序基本块指令预取技术.软件学报,2016,27(9):2426−2442. http://www.jos.org.cn/1000-9825/4859.htm

英文引用格式: Wang ED, Ni F, Chen JC, Wang HW, Tang SB. Basic-Block based instruction prefetching technology for real-time system. Ruan Jian Xue Bao/Journal of Software, 2016,27(9):2426−2442 (in Chinese). http://www.jos.org.cn/1000-9825/4859.htm

Basic-Block Based Instruction Prefetching Technology for Real-Time System

WANG En-Dong, NI Fan, CHEN Ji-Cheng, WANG Hong-Wei, TANG Shi-Bin

(State Key Laboratory of High-end Server&Storage Technology (Inspur (Beijing) Electronic Information Industry Co., Ltd), Beijing 100085, China)

Abstract: Instruction prefetching technologies proposed for general purpose computer systems cannot meet the requirements of real-time systems. One of the most important issues is that cache content pollution caused by useless prefetching loses real-time tasks' WCET estimates. And a loose on WCET analysis degrades the schedulability of the system and in turn brings down its efficiency. A basic-block based instruction prefetching method is proposed in this paper. The method performs instruction prefetching at the basic block level, avoids useless prefetching, simplifies the instruction hit/miss classifications in the worst-case execution, and reduces the WCET estimates of real-time tasks. Real-time benchmark tests show that, the method can reduce real-time tasks' WCET estimates by 20% and also improve instruction cache access performance by 10% on average.

Key words: real-time system; WCET; cache performance; instruction prefetching; basic block

在通用计算机系统中,指令 Cache 是提升系统性能的重要手段,但其在实时领域的应用面临诸多挑战.实时系统要求任务在确定的时间间隔内响应输入事件,而 Cache 访问会导致任务执行时间不确定,从而不利于任务的最坏情况执行时间(worst-case execution time,简称 WCET)分析.因此,Cache 在实时领域的应用一直受限.尽管如此,在实时系统中,应用指令 Cache 已逐渐成为趋势.一方面,实时任务的复杂性不断提高,客观上要求使用指

* 基金项目: 国家高技术研究发展计划(863)(2013AA011701)
Foundation item: National High-Tech R&D Program of China (863) (2013AA011701)
收稿时间: 2014-09-03; 修改时间: 2014-12-31; 采用时间: 2015-05-14

令 Cache 提升系统性能以满足任务执行时间要求;另一方面,指令 Cache 建模技术的不断发展,使得 WCET 静态分析技术能够将大部分的指令 Cache 访问静态归类为命中或缺失,从而降低 Cache 访问对任务执行时间确定性的影响,为指令 Cache 在实时领域的应用提供了理论基础支撑.因此,研究适用于实时系统的指令 Cache 结构新特性,具有重要的理论和现实意义.

指令预取技术是一种提升指令 Cache 访问性能的有效方式.现有的指令预取技术大多面向通用计算机系统进行设计,遵循加速常见情况的设计原则.在绝大部分情况下,这种设计原则能够提升指令访问性能.但是,实时系统最为关注的是程序在最坏情况下的执行时间,上述设计原则往往无法提供最坏执行情况下的性能保障.因此,设计面向实时系统的指令预取方法,必须考虑实时系统的特殊需求:在优化最坏执行情况指令访问性能的前提下提升平均情况指令访问性能.

本文通过分析典型实时基准测试程序的指令访问特性,结合实时任务 WCET 分析中指令 Cache 访问时间分析原理,提出了一种面向实时系统的、基于程序基本块的指令预取方法,该方法能够有效避免无效指令预取.同时,该预取方法以优化实时任务 WCET 分析为基本出发点,通过简化程序基本块内部指令访问命中情况判定降低指令 Cache 分析难度,减小了 WCET 评估值,从而提升实时系统的任务可调度性.实时基准测试程序评估结果表明:在平均执行情况下,该预取方法使得 Cache 访问缺失数降低约 40%~70%,对应的程序执行时间(时钟周期数)平均降低约 10%;在最坏执行情况下,静态缺失数降低约 50%,对应的 WCET 评估值平均降低约 20%.由此可见:该预取方法能够有效改善指令访问确定性,提升实时任务平均和最坏执行情况下的 Cache 访问性能.

1 研究背景

通用计算机系统采用指令预取技术[1−11]提升指令 Cache 访问性能.但该类预取技术仅针对平均执行情况下的指令 Cache 访问缺失数优化设计,未考虑最坏执行情况下的指令 Cache 性能.2007 年,南伊利诺伊大学(SIUC)的 Yan 及 Zhang 评估了一种典型指令预取技术(Next-N-Line 预取[12])对实时任务 WCET 评估值的影响.结果表明:Next-N-Line 预取技术无法有效降低程序 WCET 评估值,在提升最坏执行情况下的指令 Cache 性能方面效率低下.其主要原因在于:预取单元顺序预取后续 N 行而未考虑程序控制流的影响,可能导致预取指令与其他已存于指令 Cache 中的有用指令发生冲突.由于静态 WCET 分析器采用保守策略评估最坏情况下的 Cache 污染问题,即:所有可能发生冲突的指令均可被预取指令污染,导致 WCET 评估值过高,从而给实时系统的任务调度带来很大的压力.此外,预取无用指令会导致额外的能耗,对于嵌入式系统,这一点是重要的限制.Ding 等人的后续工作[13]提出了一种循环导向的预取机制及一种面向 WCET 的软件预取方法,该方法考虑了程序控制流的影响,但是并未彻底消除预取导致的指令 Cache 内容污染.

为了消除指令 Cache 访问时间的不确定性,同时提升指令 Cache 访问的性能,Lee 等人[14]基于时间模型对缓存预取模式进行了 WCET 分析,其工作可视为替代传统指令 Cache 的新型指令缓存结构设计,而非指令 Cache 结构的优化.Martin[15]提出了一种称为 Method Cache(方法高速缓存)的新型 Cache 结构,与传统的、以 Cache 行为单位的指令 Cache 不同,方法 Cache 存储整个方法.在方法 Cache 中,Cache 命中与缺失仅仅发生在方法被调用或者返回时.在该技术中,作者假设所有的分支均是相对的,且没有分支的目标位于其他方法中.在真实硬件系统中,该假设通常无法成立.故该技术具有很大的局限性,仅仅面向 Java 处理器.

总而言之,上述两方面的工作均存在一定的局限性:首先,它们并非针对实时系统的 WCET 分析需求进行设计,可能引入某些不利于 WCET 分析的副作用,如上文提及的无效预取及 Cache 干扰问题;其次,抛弃传统指令 Cache 结构的新型结构设计适用性有待验证,无法实现快速应用与普及.

因此,理想的提升实时系统指令 Cache 性能的设计应当具备如下两方面的特性.

(1) 以优化 WCET 分析为基础进行设计.最坏情况下的 Cache 性能对实时系统至关重要,因此,须以优化 WCET 分析为设计出发点.在保证 WCET 分析不受影响的前提下,优化最坏及平均执行情况的指令 Cache 性能;

(2) 易于应用和推广.最好基于现有指令 Cache 结构进行优化,而非全新设计,减小设计验证难度,实现快

速应用及推广.

2 方法动机

指令Cache相关的WCET分析难点在于确定每条指令的访问命中情况.本节以一个典型的开源静态WCET分析工具——Chronos[16]为例,通过总结WCET分析中指令访问命中情况判定难点得出面向实时系统的、优化WCET分析的指令预取方法设计要点,进而提出基于基本块的预取思想.

静态WCET分析是评估应用程序在特定硬件平台上的执行时间上限的过程,其难度取决于两个方面:(1)硬件平台的复杂性;(2)程序执行流程.对于特定程序的二进制代码,程序静态执行流程确定,但硬件平台结构差异会导致程序执行时间分析的精度及难度不同.由于WCET分析评估程序在所有执行情况下的运行时间上限,因此,获得准确WCET值的理想方式是遍历程序运行的整个空间,包括所有可能的输入及执行上下文等.在实际操作过程中,这种穷举方法通常无法实现.例如:假定一个程序具有10个无直接关联的执行分支,那么在所有可能输入情况下,仅此10个分支就会导致1 024(即2^{10})种可能的程序执行流程.另外,程序流程中的嵌套循环、函数调用以及复杂处理器微结构(流水线、Cache等)的影响,会使得穷举方法彻底失效.为此,研究人员基于数学建模思想开发WCET静态分析工具来分析程序流程及特定硬件结构对程序执行时间的影响.WCET静态分析工具得到的WCET评估值不小于真实的WCET值.评估值越小,与真实的WCET值越接近,分析的有效性越好,对系统任务可调度性分析越有利.

Chronos是由新加坡国立大学开发的开源WCET分析工具,它能够对多种处理器微结构特性(如流水线、分支预测器、指令Cache等)及其相互影响进行建模分析.接下来将简述其中的指令Cache建模及行为分析原理[16].

实时程序的最坏情况执行时间由其在最坏情况下的执行路径与该路径上的指令的执行时间共同决定.WCET分析技术使用高层分析(也称为程序路径分析)及微架构建模方法分别对二者进行分析.其中,高层分析构建程序控制流图(control flow graph,简称CFG),而微架构建模返回CFG中每个节点的执行时间上限.在Chronos中,程序WCET值的计算过程被定义为整数线性规划(integer linear programming,简称ILP)问题,其目标函数为

$$WCET = \text{Maximize}\sum_{b\in\beta} N_b \times C_b \tag{1}$$

其中,N_b表示基本块b的执行次数,作为ILP变量由ILP求解程序求解;C_b代表基本块b的WCET评估值常量;β表示程序基本块集合.

在Chronos中,高层分析过程为程序中的每个函数创建一个局部控制流图(local CFG),随后,根据函数间的调用关系构建全局控制流图(global CFG).局部及全局CFG中的每个节点代表一个程序基本块,程序基本块(basic-block)是具有惟一执行入口和出口的代码片段.基本块中的指令按序执行,一旦一条指令投入运行,该基本块中的后续指令都将被执行.对于给定的Cache配置,一个基本块可覆盖一个或多个存储块,其中,每个存储块映射到一个Cache行.假设B_i表示一个基本块,$B_{i.1}$,$B_{i.2}$,..., $B_{i.n_i}$分别表示基本块所包含的n_i个存储块.为了计算基本块B_i的WCET评估值,必须对B_i中的所有指令进行访问命中/缺失情况分类.B_i中被静态确定为命中的指令数越多,基本块B_i的WCET评估值越小,进而使得整个程序的WCET评估值越小.对于任意给定存储块$B_{i.k}$,指令访问缺失仅可能发生在访问存储块第1条指令时.因此,根据基本块中的每个存储块的第1条指令的命中情况,即可获得基本块B_i的访问命中情况.

上述过程分析存储块中的空间局部性.除此以外,Chronos还分析指令访问序列中的时间局部性.Chronos以循环(loop)为单位跟踪存储块的重用信息,以确定在其两次连续访问间是否存在冲突.在构建局部CFG过程中,确定每个基本块所在的循环层次,并记录相应的循环编号(最外层循环编号为0).在一层循环中,若两个存储块映射至同一Cache组,则定义一次访问冲突(conflict).当一个存储块所在Cache组的冲突次数大于或者等于指令Cache组相联度时,对该存储块中第1条指令的再次访问判定为缺失,否则归类为命中.此外,每个基本块与一个

称为命中循环(hit loop)的循环关联.循环 i 被定义为存储块 mb 的命中循环当且仅当在循环 i 中,mb 所在 Cache 组的冲突次数小于 Cache 组相联度;而在循环 i 的直接外层循环中,mb 所在的 Cache 组的冲突次数大于或等于相联度.假若某一存储块不存在满足上述条件的循环体,则该存储块的命中循环为空(null).因此,在存储块 mb 的命中循环及其内层循环,除去第 1 次访问,后续所有对该存储块的访问都将命中,而在命中循环外部对 mb 所在 Cache 组的再次访问都判定为访问缺失.所有这些命中与缺失信息,连同 CFG 确定的分支调用关系,都作为常量输入 ILP 求解程序.

为了获得安全、精确的 WCET 评估值并对流水线行为进行建模,在计算基本块 B_i 的执行时间上限时,Chronos 考虑基本块执行上下文.位于基本块 B_i 之前及之后的若干条影响 B_i 执行时间的指令构成了 B_i 的执行上下文,分别称为序言(prologue)及结语(epilogue).为了评估安全,Chronos 假设位于序言和结语中的指令的访问情况未知,因此在流水线分析中,取指阶段需要考虑命中与缺失两种情况,加大了时间分析难度.

从上述指令 Cache 命中情况分析过程可知,指令 Cache 访问时间分析难度在于判定程序基本块中的每个存储块的命中与缺失情况.静态判定为命中的存储块的数目多少,将决定整个程序 WCET 评估值的大小.因此,假若使用某种机制能够保证每次对一个基本块的访问至多导致一次访问缺失,同时不引入额外的缺失,则可极大地简化 WCET 分析过程.对于 CFG 中的每个节点,只需执行一次分析即可得到整个基本块的某次访问的命中指令数,同时降低静态指令访问缺失数.同时,序言与结语中的指令访问更易于归类,从而简化上下文相关分析,有利于实现精确的 WCET 评估.

因此,本文提出一种基于程序基本块的指令预取方法(basic-block based instruction prefetching,简称 BBIP).在该预取方法中,当访问一条指令发生缺失时,除了将当前指令所在的存储块调入指令 Cache 以外,还将位于同一基本块的后续存储块也调入指令 Cache 中,从而避免了基本块中的后续存储块指令访问缺失.

3 BBIP 预取机制收益定量分析

接下来将结合静态 WCET 分析中指令 Cache 访问命中与缺失判断的基本原理,定量分析 BBIP 预取方法的指令 Cache 访问性能收益.分析将从指令访问时间确定性及平均指令访问性能两方面展开.

3.1 指令访问确定性收益分析

由公式(1)可知,程序的 WCET 评估值由程序中每个基本块的 WCET 评估值及其执行次数共同决定.其中:基本块的执行次数主要依赖于程序执行流程,由程序本身属性及输入共同决定,几乎独立于处理器底层微架构,因此不受指令预取方法影响;而基本块的 WCET 评估值依赖于底层微架构,是预取方法主要的优化对象.

基本块 b 的 WCET 评估值可由公式(2)计算得出:

$$C_b = \sum_{b_i \in b} T_{b_i} \tag{2}$$

其中,b_i 表示基本块 b 中的一条指令,T_{b_i} 表示指令 b_i 的执行时间上限评估值.该上限值的计算在一定的执行上下文中进行,执行上下文已将处理器微结构(如流水线、Cache 层次等)的影响考虑在内.根据 b_i 是否在指令 Cache 中命中,T_{b_i} 可按照公式(3)计算:

$$T_{b_i} = \begin{cases} Lat_{hit}, & \text{指令}Cache\text{访问命中} \\ Lat_{hit} + Cost_{miss}, & \text{指令}Cache\text{访问缺失} \end{cases} \tag{3}$$

其中,Lat_{hit} 表示指令访问命中时的指令执行时间;$Cost_{miss}$ 表示指令访问缺失时,从下级存储器中调入存储块并执行存储块替换花费的额外时间.注意:对于支持多发射乱序执行流水线的处理器,由于 Cache 命中与缺失可能影响程序执行的其他流水线阶段,上述公式尚不够精确,但这种影响非常有限,通常认为其可忽略.

在传统的不具备预取功能的指令 Cache 中,在计算基本块的 WCET 评估值时,静态判定为 always-hit(总是命中)的指令被视为访问命中.指令被静态 WCET 分析方法判定为总是命中,意味着在程序的实际执行过程中,无论输入导致程序执行流程如何变化,该条指令的访问总能够在指令 Cache 中命中.例如:如果若干条顺序执行的指令(I_1,I_2,…,I_n)映射到同一 Cache 行,则除了第 1 条指令(I_1)可能发生指令访问缺失外,后续的其他指令(I_2,…,

I_n)的访问总能命中.这是因为:指令以 Cache 行为单位调入指令 Cache,若 I_1 指令访问不命中,则整个 Cache 行被调入指令 Cache;否则,若 I_1 访问命中,则表示 Cache 行已经存在于 Cache 中.无论上述哪种情况,指令 $I_2,\ldots,I_n$ 的访问均将在指令 Cache 中命中.不能静态判定为总是命中的指令的命中情况借助于其他方式进行判定,如分析位于同一循环内的指令映射至同一 Cache 组的访问冲突次数.

在基本块的 WCET 分析过程中会出现如下情况:当基本块中的大部分指令被划分为命中时,其中一些指令由于缺乏总是命中的保障不得不归为缺失或者未知,从而导致 WCET 分析难度增加,WCET 评估值变大.基于基本块的指令预取技术设计恰好提供这种保证,即:位于同一基本块中的指令的一次执行最多只能引入一次访问缺失.这种处理方式简化基本块的 WCET 分析,降低 WCET 评估值.同时,基本块执行上下文(序言和结语)中的指令命中情况判定更明确,进一步简化了 WCET 分析.

接下来考虑如下两种场景:

(1) 无指令预取场景(no instruction prefetching scene,简称 NIPS):假设程序运行于不支持指令预取功能的处理器上.同时,一个 Cache 行占 B 个字节,基本块 b 位于程序最坏执行路径(WCEP)上,其包含 C_b 条指令,在程序最坏执行情况下被执行 N_b 次;

(2) 基于基本块指令预取场景(BBIP scene,简称 BBIPS):程序运行于支持基本块指令预取的系统之上,系统其他配置相同.

在上述两种场景中,基本块在 Cache 中占据$(C_b+\tilde{B}_1)/B$ 个 Cache 行.假设没有程序执行的其他信息来保证对于一个 Cache 行的访问会导致命中,那么在 NIPS 中,被 WCET 分析过程归类为缺失的指令访问数为$(C_b+\tilde{B}_1)/B\times N_b$.而在 BBIPS 中,只有 N_b 次访问被归类为缺失.

依据实时基准测试程序基本块特性,作出如下假设分析:基本块 b 覆盖 3 个 Cache 行,在最坏执行过程中被执行 100 次,那么在针对基本块 b 的 WCET 分析过程中,200 条在 NIPS 中被归类为缺失的指令访问在 BBIPS 中转化为命中.对于指令 Cache 的 WCET 分析,指令访问缺失次数的减少意味着 WCET 评估值降低.值得注意的是:由于其他信息(如程序执行流程)的存在,在实际 WCET 分析过程中,BBIP 能够转化为命中的指令往往低于上述理想分析值.

另一方面,基于基本块的指令预取机制的使用导致在一个基本块的一次执行过程中至多存在一次指令访问缺失.一旦基本块中存在一条指令已被归类为访问缺失,则该基本块的其他指令的访问都归类为命中,该基本块的指令访问命中判断分析过程即可结束.因此,该特性能够极大地简化基本块的 WCET 分析过程.

3.2 指令访问平均性能分析

除了能够降低程序的 WCET 评估值外,BBIP 预取方法还可提升程序在平均情况下的指令 Cache 访问性能.本节后续讨论基于如下假设:

1) 程序运行于 RISC 架构的处理器之上,每条机器指令占用 8 字节;
2) 指令 Cache 的大小为 2K 字节,Cache 行大小为 32 字节;
3) 平均每 1000 次指令 Cache 访问,产生 5 次访问缺失;
4) 每次指令访问缺失导致一次内存访问;
5) 每个基本块不超过 2K 字节,因此能够完全容纳在指令 Cache 中.

显然,只有当基本块大小超过一个 Cache 行时,预取机制才能对基本块内的剩余存储块执行预取.为了简化分析,假设所有的基本块大小都不超过 256 字节(对于绝大部分实时基准测试程序,该假设成立),并且程序执行过程中对所有基本块的访问平均分布.后续讨论过程中,以 Cache 行长度为基本单位表示基本块长度,例如基本块长度为 2 表示其覆盖 2 个 Cache 行.

不使用指令预取时,对任意基本块的每个存储块的第 1 条指令的首次访问将导致一次缺失(称为冷启动缺失).由于假设程序执行对所有基本块的访问遵循平均分布,对程序中所有基本块的访问将导致平均 4.5 次缺失(见表 1).而当指令 Cache 使用基于基本块的指令预取时,不论基本块大小,只存在一次冷启动缺失.

Table 1 Cold-start misses of basic blocks with different size in NIPS and BBIPS

表 1 不同长度基本块在 NIPS 与 BBIPS 下对应的冷启动缺失次数

基本块字节数	<=32	<=64	<=96	<=128	<=160	<=192	<=224	<=256	平均
NIPS	1	2	3	4	5	6	7	8	4.5
BBIPS	1	1	1	1	1	1	1	1	1

使用从内存加载所有缺失存储块到指令 Cache 的延时来评估平均执行情况下的指令 Cache 访问性能.假设主存工作在突发(burst)模式下,访问主存的时间由加载第 1 个数据块(chunk)的延迟(记为 Lat_{fmc}),后续数据块延时(记为 Lat_{rmc})及内存总线宽度(记为 W_{mb})三者共同决定.由于总线每次只能传输一个总线宽度的数据,所以在 burst 模式中,当存储块容量(等于 Cache 行大小)小于或等于总线宽度时,一次总线传输即可将 Cache 行调入,对应延迟为 Lat_{fmc};当存储块容量大于总线宽度时,一个存储块的调入需要经过若干次连续的总线传输,第 1 次传输延迟为 Lat_{fmc},后续每次传输延迟为 Lat_{rmc}(Lat_{fmc}>Lat_{rmc}).当存储块容量非总线宽度的整数倍时,需要对存储块容量向上取整以计算传输次数.需要注意的是:本文公式中使用的除法符号均表示整数除法,舍弃计算结果的小数部分.为了简化公式表达,假设存储块长度等于内存总线宽度的整数倍.

从内存加载一个缺失存储块至指令 Cache 的时间开销,可依据下述公式计算:

$$T_{lb} = Lat_{fmc} + \left(\frac{B}{W_{mb}} - 1\right) \times Lat_{rmc} \tag{4}$$

其中,B 为存储块长度(字节数).不使用指令预取时,每次指令 Cache 访问缺失时,仅将缺失的存储块调入指令 Cache 中.因此,若基本块长度为 C,则经过 C 次独立存储访问(每次传输一个存储块长度)才能将其调入 Cache.加载长度为 C 的基本块 b 的所有存储块的时间开销可表示为

$$T_{NIP} = C \times \left(Lat_{fmc} + \left(\frac{B}{W_{mb}} - 1\right) \times Lat_{rmc}\right) \tag{5}$$

当使用 BBIP 预取方法时,每次指令访问缺失将该指令所在基本块的全部存储块调入指令 Cache.因此,整个基本块的加载只需发起一次存储访问(传输一个基本块长度)即可完成.加载基本块的时间开销可表示为

$$T_{BBIP} = Lat_{fmc} + \left(\frac{C \times B}{W_{mb}} - 1\right) \times Lat_{rmc} \tag{6}$$

因此,使用 BBIP 预取机制加载一个基本块可节省的周期数为

$$T_{diff\_b} = T_{NIP} - T_{BBIP} = (C-1) \times (Lat_{fmc} - Lat_{rmc}) \tag{7}$$

对于一个执行 N_b 次、平均缺失率为 R_b 的基本块,使用预取机制可减少的指令访问周期数为

$$T_{diff\_all\_b} = (C-1) \times (Lat_{fmc} - Lat_{rmc}) \times N_b \times R_b \tag{8}$$

由公式(8)可知,BBIP 预取技术对于平均 Cache 性能的优化程度由基本块长度(由基本块长度与指令 Cache 的行大小共同决定)、内存访问数据块延迟及执行路径上基本块的缺失次数共同决定.在存储系统参数(Cache 行大小、Lat_{fmc}、Lat_{rmc})一定时,基本块长度越大,基本块缺失次数越多,使用 BBIP 预取技术带来的指令访问性能提升效果越显著.另外,当其他参数一定,Cache 行大小变化时,同一基本块覆盖的 Cache 行数 C 变化.Cache 行容量越小,相同的基本块对应的 C 越大,相应的预取时间收益越大;反之,Cache 行容量越大,同一基本块对应的 C 越小,预取收益越小.一种极限情况是:当系统中所有的基本块长度均不超过一个 Cache 行(即 C=1)时,指令访问过程不执行指令预取,无预取收益.

结合上述分析可知:基于基本块的指令预取技术一方面可简化 WCET 分析,降低 WCET 评估值,从而改进实时系统的任务可调度性;另一方面,还可提升 Cache 访问性能,并可降低因频繁访问内存引入的能耗开销.

4 BBIP 预取机制软硬件设计

4.1 预取机制描述

在基于基本块的指令预取技术中,当一条指令访问发生缺失时,该指令所在基本块的所有后续指令都被提

前加载到指令 Cache 中,从而避免该基本块后续存储块访问缺失.由于基本块的大小不固定,因此需要使用一定的软件方法或硬件部件记录指令基本块信息(起始地址和长度),以指导后续的指令预取过程.在通用计算机系统中,由于系统中运行的应用程序时刻变化,因此除非采用一些特殊的处理方法(例如采用定制的编译器在编译时收集相关信息)否则往往难以预知应用程序的基本块信息;相反,实时系统则不存在这种阻碍.运行于特定实时系统(特别是硬实时系统)中的应用程序,往往在系统设计时即已确定.不仅如此,系统的目标任务集行为数据还常被用于指导及优化硬件结构(例如流水线及 Cache 层次等)的设计.由此可见,收集程序基本块信息以指导指令预取在实时系统中是可行的.

4.2 硬件实现

为了支持基本块层次的指令预取,必须记录程序的基本块信息.在实时系统中,一个应用程序的功能通常较为单一,因此程序规模相对较小.这意味着程序的基本块集合较小.而且基本块的长度通常较小,例如本文评估过程使用的大多数基准测试程序中的基本块长度不超过 32 条指令,一般在 16 条指令以下,占据两到 3 个存储块.表 2 列举了一些基准测试程序的基本块分布情况.从中可知:绝大部分基准测试程序包含的基本块数量较少(adpcm 除外),而且基本块包含的指令数不超过 16.假设每个 Cache 行可容纳 4 条指令,则绝大部分基本块的长度不超过 4 个 Cache 行.

Table 2 Distributions of basic blocks in different benchmarks

表 2 基准测试程序基本块分布情况

指令数	[1,2]	[3,4]	[5,8]	[9,16]	>16	总数
adpcm	49	18	38	28	10	143
cnt	6	8	7	0	1	22
crc	12	6	2	7	1	28
edn	14	8	11	13	7	53
*fft*1	15	21	10	9	2	57
fir	9	4	0	4	0	17
lms	20	6	15	16	1	58
matmult	6	5	8	2	1	22
qurt	13	3	5	7	1	29

为了保存基本块信息,构造一个称为基本块信息表(basic-block based information table,简称 BBIT)的硬件结构,见表 3.BBIT 每个表项包含两个字段:基本块的起始地址(BBA)及基本块的长度(BBS).由于程序指令以 Cache 行为单位从主存中加载,因此,基本块信息(BBA 及 BBS)也可对齐到 Cache 行.BBA 记录基本块第 1 条指令的地址对应的块地址,而 BBS 记录基本块跨越的 Cache 行数.例如:当指令 Cache 的行大小为 32 字节时,一个起始自 0x400128,具有 48 字节长的基本块对应的 BBA 及 BBS 分别为 0*x*400120 及 2.采用基于 Cache 行的对齐方式可节省基本块信息的存储开销.

Table 3 A BBIT example with 100 items

表 3 包含 100 表项的 BBIT 示意图

起始块地址	块长度
BBA_1	BBS_1
BBA_2	BBS_2
…	…
BBA_i	BBS_i
…	…
BBA_{100}	BBS_{100}

访问支持 BBIP 预取功能的指令 Cache 的过程包括如下几个步骤.

1. 地址分发

程序执行过程发起一次指令访问,该指令对应地址(称为目标地址)被送往指令 Cache.根据指令 Cache 的配置(组数、块大小),该地址被分为 3 个字段:标签(tag)、组号(set)以及块内偏移(offset).其中,tag 用于地址查找时的匹配比较,set 用于定位地址所在的 Cache 组,offset 用于从定位的 Cache 行中取出对应的指令字.此过程与访

问传统的指令 Cache 过程一致.

2. 查找

与传统指令 Cache 相比,访问支持 BBIP 预取的指令 Cache 在查找 tag 表进行 tag 比较的同时,需要执行 BBIT 查找.具体的查找过程可分为两个并行执行的子过程.

1) tag 查找:使用目标地址的 set 字段检索 tag 表定位目标地址所映射到的目标组,然后,将目标地址的 tag 字段与该组中所存储的 tag 值逐一比较.假若存在一个匹配项,使用 offset 从对应的 Cache 数据行中取出对应的指令字,根据替换算法更新 Cache 状态,最后返回指令字;否则,本次 Cache 访问缺失,从 Cache 层次的下层中取回相应的 Cache 行并更新 Cache 状态;

2) BBIT 查找:将目标地址的块地址(由 tag 与 set 字段拼接而成)与 BBIT 的 BBA 字段比较:若存在一个匹配项,则返回该匹配 BBA 对应的 BBS 字段;否则,返回一个非法的 BBS(如 0 或 1),指示不执行预取.

3. 预取

假若 tag 查找过程没有找到对应的 Cache 行,并且 BBIT 查找返回了合理的 BBS 字段,则对目标地址所在的基本块的后续存储块(BBA+1,...,BBA+BBS–1)执行预取.

由上述指令 Cache 访问过程可知,只有当指令访问缺失(即,tag 查找没有发现匹配项)时才执行基本块预取.对于绝大部分实时应用程序,指令 Cache 访问缺失率不高.例如本文使用的基准测试程序,缺失率均小于 10%(见表 4),因此预取操作发生的频度较低.另外,如表 2 所示的基本块通常较小(如覆盖 2 或 3 个存储块).因此,对于具有多个读写接口的非阻塞指令 Cache,预取操作可与当前基本块中已取回的指令的执行部分或全部重叠,预取操作的时间开销可极大地降低.此外,在支持突发访问(burst)模式的存储器中,对位于同一基本块内的多个存储块的连续访问,可能因为其在存储器中处于相同的行而使访问延迟降低.

Table 4 Miss-rates of benchmarks

表 4 基准测试程序缺失率

程序	访问数	缺失数	缺失率(%)	程序	访问数	缺失数	缺失率(%)	程序	访问数	缺失数	缺失率(%)
adpcm	124 977	562	0.45	*edn*	63 756	816	1.28	*lms*	219 199	7405	3.38
cnt	4 071	25	0.61	*fft*1	1 511	44	2.91	*matmult*	133 668	29	0.02
crc	21 464	34	0.16	*fir*	251 090	18	0.01	*qurt*	638	58	9.09

在硬件实现的、支持基本块预取(BBIP)的指令 Cache 中,BBIT 硬件结构的实现是关键.BBIT 硬件结构的一种直观设计(如图 1 所示)是使用一个支持二分查找的有序列表,从而有效降低 BBIT 查找过程中的比较次数,降低延迟.但是处理器芯片空间资源宝贵,而支持二分查找的硬件列表的实现代价昂贵,因此,必须采用更为高效的方法设计 BBIT 结构.本文提出了一种更为有效的设计方法,称为 BBIT Cache.BBIT Cache 的硬件结构(如图 2 所示)与通用 Cache 类似.在 BBIT Cache 中,基本块起始块地址 BBA(或若干高位)作为标记(称为 BBA tag),基本块大小 BBS 作为数据.由于不同基本块的 BBA 不同,因此使用 BBA 的若干低位将它们映射到不同的 Cache 组,此外,使用组相联方式来减少不同基本块映射到同一 Cache 块引起的冲突.每次进行 BBIT 查找时,将目标地址的基本块地址 BBA 中的若干低位作为关键字定位该基本块信息所在的组,然后将该组中保存的 BBA tag 与目标地址的 BBA 比较,如果匹配,返回对应的 BBS,否则 BBIT 查找返回失败.

BBIP 预取机制作为传统指令 Cache 的一种附属特性存在(如图 1 所示),通过禁用 BBIT 查找,可禁止 BBIP 预取,使得指令 Cache 访问过程与传统指令 Cache 一致.在一些特殊情况下,此种特性保证指令 Cache 能够正常工作.例如在多任务实时系统中,某个实时任务的基本块尺寸可能过大,以致不能记录在 BBIT(记录 BBS 的位数有限)中,此时,可通过暂时禁止 BBIP 预取保证该任务正常使用指令 Cache.当该任务让出处理器,其他任务投入运行时,可以重新开启 BBIP 预取.上述使用场景意味着支持 BBIP 预取的指令 Cache 可提供有差别的指令访问性能,满足不同系统运行时需求,如能耗需求.另外,通过固定 BBIT 的大小,如 32 项,可以依据某种特定的基本块选取算法选择加载有价值的基本块存于 BBIT,从而在保证一定的硬件开销的情况下提供最优的预取性能.

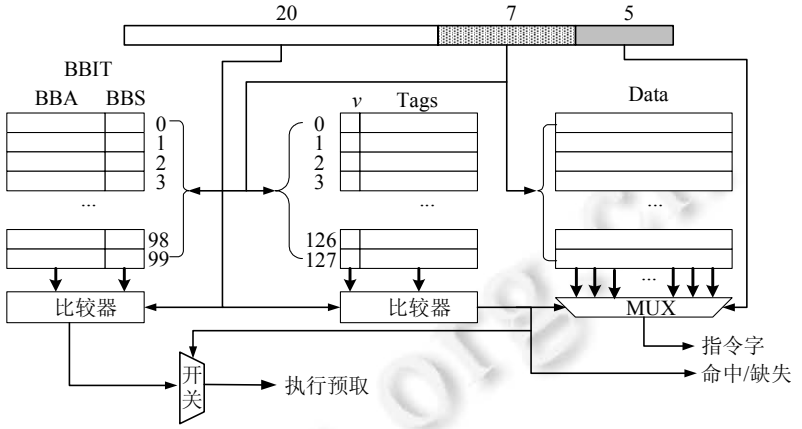


Fig.1 Structure of a directly-mapped 4K instruction cache with 32-byte blocks and BBIP enabled

图 1 支持 BBIP 预取的、容量为 4K、块大小为 32 字节、直接映射的指令 Cache 示意图

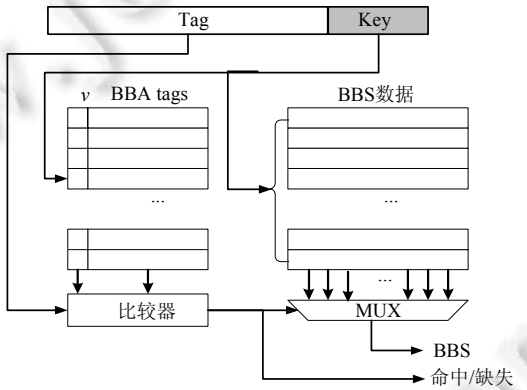


Fig.2 An example of BBIP cache

图 2 BBIT Cache 示意图

4.3 软硬件协同实现

除了硬件方式外,也可在现代处理器提供的硬件特性的辅助下使用软件方式实现BBIP指令预取功能.现代微处理器通常提供一些非阻塞的软件预取指令,这些指令用于将某些(由特定预取算法决定的)Cache 行提前调入指令 Cache,从而避免后续指令访问缺失.此类指令通常使用指令字的一些位记录预取所需信息(如预取距离).如上文所述,实时应用的绝大部分基本块的大小不超过 16 个存储块.因此,可使用预取指令的若干位(如 4 位)记录一个基本块的长度以指导预取.此外,对于没有提供预取指令的 ISA,也可使用一些替代方案.SimpleScalar 工具链提供了一种称为 PISA 的类 MIPS 指令集,该指令集中每条指令占用 8 字节,指令字中的若干位没有使用或预留用于将来的拓展.对于此类指令集,可在不增加额外指令的情况下,利用指令字中的这些保留位来编码预取信息(prefetching instruction field,简称 PIF).除了 ISA 的支持外,预取机制实现方式还需编译器的协助.在程序编译过程中,编译器除了完成常规的编译链接工作之外,还需完成如下工作.

(1) 分析程序的控制流程,构建程序控制流图;

(2) 收集基本块信息,如:基本块入口地址及长度;

(3) 对每个基本块的入口指令的 PIF 域进行编码,同时记录每个基本块的长度.

在程序执行过程中,一旦指令访问发生缺失,且指令的 PIF 域记录了预取相关信息时,则依据预取信息执行基于基本块的预取.此种实现机制能够方便地处理一些特殊情况,如基本块尺寸过大导致预留域无法进行预取信息编码,此时,编译器仅仅需要跳过这些基本块,不对其进行预取信息编码.在程序执行过程中,对这些没有编码记录的基本块不执行预取,按照传统指令 Cache 访问的方式处理.

对于本身支持指令预取的 ISA,编译器需要完成的工作相对简单.首先,对程序执行路径进行分析来收集所有的基本块的入口地址及长度,此过程需底层硬件信息支持,如程序运行硬件平台的 Cache 配置信息;接着,对程序执行静态 Cache 分析(如文献[18]),将对基本块的指令访问分为总是缺失(always-miss)、总是命中(always-hit)、首次访问缺失(first-miss)及首次访问命中(first-hit);随后,编译器对归类为 always-miss 的基本块的第 1 条指令进行预取信息编码或将其替换为一个对预取例程的调用.预取例程是一小段调用预取指令执行基本块预取的指令序列.由于预取指令仅仅插入基本块内部,因此不会对程序执行控制流程产生影响.为了避免预取例程对指令 Cache 可能造成的干扰,将其置于一个特殊的程序存储段,对位于该存储段中的指令的访问将绕过指令 Cache,从而避免了对指令 Cache 的污染.

对于本身不支持指令预取的 ISA,也可采用与上述类似的方法实现基于基本块的指令预取.在执行预取的子例程中,使用加载指令的副作用将需要预取的存储块提前加载至指令 Cache 中.表 5 展示了一段使用加载指令模拟预取的伪代码.

Table 5 Pseudocode to mimic prefetching with load instructions

表 5 加载指令实现的模拟预取伪代码

```
BBA:基本块第 1 条指令地址对应块地址;
BBS:基本块占据的 Cache 行数;
prefetch_mb=BBA+sizeofMb;   //sizeofMb 为 Cache 行容量
当(prefetch_mb<BBA+BBS*sizeofMb),执行如下操作:
{
使用加载(load)指令读取位于 prefetch_mb 处的指令;丢弃;
Prefetch_mb+=sizeofMb;
}
```

与硬件实现方式相比,软件实现方式更灵活,但由于需要执行额外的指令片段,效率较硬件实现方式低.

4.4 预取机制的拓展与优化

在 BBIP 预取机制硬件实现中,BBIT 硬件结构设计至关重要.接下来,首先阐述如何在多任务上下文环境中操作 BBIT 以有效实现指令预取;随后讨论当 BBIT 容量不足时,如何选择最优基本块以提供最小的 WCET 评估值.

由于实时系统的任务集在系统设计时即已确定,因此所有任务的基本块信息均可提前获取.假设 BBIT 硬件表包含 S 个表项,系统任务集 T 包含 n 个任务 $t_1,t_2,...,t_n$.任务 $t_i(i\in[1,n])$对应的基本块集为β_i,则系统的基本块集合$\theta=\{\beta_1,\beta_2,\ldots,\beta_n\}$.根据基本块集合$|\theta|$与 S 的大小关系不同,可按照如下方式组织 BBIT 内容:

(1) $|\theta|<=S$,在系统启动时,将θ中长度大于 1 的基本块信息装入 BBIT;

(2) 对于任意 $i\in[1,n]$,当$|\beta_i|<=S$ 时,在任务切换时将投入运行任务的长度大于 1 的基本块信息装入 BBIT;

(3) 对于某些任务 t_i,$|\beta_i|>S$,选取任务基本块集合中的部分存储 BBIT 中.

对于上述后两种情况,需要权衡多次加载 BBIT 的时间开销与预取收益的关系.当多次加载 BBIT 引入的时间开销小于预取带来的性能收益时,任务切换时加载 BBIT 才有意义;否则,在系统启动时,从θ中选取最优的基本块一次装入 BBIT 将带来更好的性能.一般而言:当 S 较大(如 128)且任务集包含的任务数 n 较小(如 8)时,系统启动时一次装入往往优于每次任务切换时装入基本块信息至 BBIT 中;否则,随任务切换多次装入 BBIT 更优.

由于处理器片上空间有限且代价昂贵,使用大量片上资源存储基本块预取信息往往得不偿失.这意味着:BBIT 的容量通常有限,只有保存部分基本块信息.此外,在程序执行过程中,不同基本块具有不同的执行频度,基本块对程序执行时间的贡献与其执行频度及包含的指令数呈正相关关系.因此,对执行频度低、包含指令少的

基本块执行 BBIP 预取的收益小于执行频度大、包含指令多的基本块.因此,选取预取收益大的基本块信息存入容量有限的 BBIT 对 BBIP 预取性能至关重要.

本文给出一种直观的、基于贪心算法思想的方法用于选取合适基本块填充 BBIT.该算法基于执行频度选取最优基本块,算法流程如下.

(1) 硬件配置初始化.具体而言,对影响 WCET 分析的处理器微结构(主要是高速缓存、流水线及分支预测器等)的参数进行设置;

(2) 通过编译器或者静态程序特征分析工具收集静态程序基本块集合;

(3) 从静态程序基本块集合中剔除只覆盖一个存储块及长度大于 Cache 组数的基本块;

(4) 使用 WCET 分析工具对任务执行 WCET 静态分析,收集最坏执行情况下的基本块的执行频度,并将每个程序基本块表示为一个三元组:(起始块地址,块长度,执行频度);

(5) 按照基本块的预取收益(定义为:执行频度×(基本块覆盖的 Cache 行数−1))从高到低的顺序对静态程序基本块集合中的基本块进行排序,得到一个有序的基本块集合;对于预取收益相同的基本块,依照块长度从大到小的顺序排列;预取收益和块长度均一致的基本块的排列顺序随机;

(6) 假如 BBIT 包含 n 个表项,将有序基本块集合中的前 n 个基本块的信息存储到 BBIT 中.

5 实验评估与结果分析

本节将从平均及最坏执行情况下的 Cache 访问性能两个方面评估 BBIP 预取机制的效果.

5.1 平均执行情况下的Cache访问性能

虽然实时系统最为关注程序最坏执行情况下的性能及运行时间,但是,提升程序在一般(平均)运行情况下的性能从而缩短其运行时间同样具有重要的现实意义.实时系统通常为嵌入式系统.与桌面通用计算机系统不同,嵌入式系统通常具有较小的体积,对功耗具有更高的要求.此外,许多嵌入式系统是便携式设备,没有显式外接电源,使用电池供电.提升系统运行时 Cache 访问性能等同于提升程序的执行性能,缩短程序的运行时间.当系统的任务负载一定时,任务的提早完成通常意味着系统功耗的降低.

本文使用一个轨迹驱动(trace-driven)、时钟精确的模拟模型——STrace 评估使用 BBIP 后实时任务的 Cache 访问性能.STrace 以 SimpleScalar 时序模拟模型为原型开发.模拟过程所使用的输入轨迹使用定制的 sim-cache 模型从 SNU 实时基准测试程序[19]提取.通过测量每个基准测试程序在不使用预取及使用 BBIP 预取(分别对应 NIPS 及 BBIPS 场景)时的程序模拟运行周期数(记为 SC)及指令 Cache 访问缺失数(记为 CMC)来评估 BBIP 指令预取技术对于 Cache 访问性能的改进效果.我们假设:在 BBIPS 场景中,所有长度大于 1 的基本块的缺失导致指令预取,缺失基本块的所有后续存储块都被提前加载到指令 Cache 中.这意味着,BBIPS 场景中的基本块的一次访问至多导致一次指令 Cache 访问缺失.

评估过程使用的 Cache 及存储配置见表 6.

Table 6 Configurations of instruction cache and main memory

表 6 指令 Cache 及主存主要参数配置

配置项	配置值	配置项	配置值
Cache 容量(字节)	2K, 4K, 8K, 16K, 32K, 64K	主存访问模式	突发式访问
Cache 块大小(字节)	8, 16, 32, 64, 128, 256	第 1 数据块传输延迟(周期)	18
相联度	直接映射	后续数据块传输延迟(周期)	2
替换策略	LRU	存储总线宽度(字节)	8

对于每种不同的 Cache 容量及块大小,测量每个基准测试程序在 NIPS 及 BBIPS 两种场景下的 SC 及 CMC,并使用相对偏差衡量测试量的改进程度.测试分量 X 的相对偏差定义为

$$RB(X)=\frac{x-x'}{x} \tag{9}$$

其中,x 为测量基准值,x'为测量值.$RB(X)>0$,表明测量值优于基准值;否则,测量值与基准值相当或次于基准值.后续讨论分别使用 NIPS(BBIPS)场景下对应的 SC 及 CMC 值作为评估量的基准值(测量值),来评估 BBIP 预取机制在提升平均情况下的 Cache 访问性能方面的效果.

需要注意的是:SC 表示程序所有(命中与缺失)指令而非仅仅发生缺失的指令对应的模拟周期数,这样能够更为直观地说明 BBIP 预取机制对于程序执行过程中指令访问部分的加速效果.但是由于指令访问周期数由命中及缺失两部分构成,当指令 Cache 访问命中时间占主导地位时,SC 的相对偏差可能不如仅记录缺失指令访问周期数相对偏差显著.

评估结果表明:对于特定的 Cache 容量,不同 Cache 行大小对应的 SC 及 CMC 相对偏差具有类似趋势.因此,此处以 4K 字节容量为例进行说明.表 7 中随着 Cache 行大小从 256 字节减小为 8 字节,SC 的相对偏差增大.这种趋势出现的原因为:Cache 行大小越大,块内指令间的空间局部性利用越充分.同时,对于特定的基本块, Cache 行大小越大,该基本块覆盖的 Cache 行越少,被预取的存储块数越少,能够避免的 Cache 缺失数越少.当 Cache 行大小能够容纳下程序中最大的基本块时,预取失效.此时,NIPS 与 BBIPS 等效,相对偏差为 0.当 Cache 行为 32 字节时,使用 BBIP 预取能够使得测试程序的模拟周期数平均下降约 11.9%;而当使用 8 字节大小的 Cache 行时,则模拟周期数可下降约 40%.

Table 7 Relative bias of simulation cycle (SC) for different cache block size (the cache size is 4K bytes)

表 7 不同 Cache 行大小对应的模拟周期(SC)的相对偏差(Cache 容量固定为 4K 字节)

RB(SC)	256B	128B	64B	32B	16B	8B
adpcm	–0.015	0.001	0.058	0.144	0.31	0.5
crc	0	0.009	0.027	0.075	0.176	0.329
*fft*1	0.003	0.01	0.054	0.171	0.344	0.555
fir	0.001	0.01	0.031	0.107	0.237	0.42
lms	0	0.001	0.004	0.014	0.035	0.077
matmul	–0.001	0.007	0.057	0.151	0.323	0.53
qurt	–0.001	0.01	0.06	0.171	0.358	0.571

公式(7)也可解释这一趋势.由于 $Lat_{fmc}>Lat_{rmc}$,因此,BBIP 相对于 NIP 节省的时间随着 Cache 行大小的增大下降,反之亦然.而公式(8)表明:对于特定的一个基本块,使用 BBIP 预取技术能够节省的模拟周期数与不使用预取时该基本块的缺失次数成正比.因此,SC 的相对偏差与程序的缺失率正相关.这解释了在表 7 中不同程序的相对偏差变化快慢不同的原因.其中,代表 *fft*1 的变化最平滑,而 *qurt* 变化最剧烈.

需要注意的是:Cache 行大小为 256 字节时,程序的 SC 的相对偏差接近于 0;甚至对于某些程序,其值略小于 0.这是因为评估使用的基准测试程序中,几乎所有的基本块均小于 256 字节,使用 256 字节大小的 Cache 行使得预取操作几乎不执行.在评估过程中,我们使用简化的预取过程.当对一个程序基本块的访问发生缺失时,不论基本块的后续存储块是否已在指令 Cache 中,均对所有后续存储块执行预取操作.根据突发式存储访问模型,这种对已经存在于指令 Cache 的存储块执行预取,会导致不必要的访问延迟.而在 NIPS 场景中,这种延迟不存在.在真实硬件系统中,这种由于假缺失导致的访问延迟,可通过一些简单的检测机制加以避免.

在表 7 中,SC 相对偏差随着 Cache 行大小的减小,增长速度增加.两个因素解释了这一现象.

- 首先,当 Cache 行较小时,缺失率的增加较快,从而导致 Cache 缺失次数的增加较快.根据公式(8)可知:节省的 Cache 访问时间增加较快,从而导致相对偏差变化较快;
- 其次,由于大部分的基本块较小,小于 128 字节,因此当 Cache 行较大时,很少的指令访问从 BBIP 预取中获益.

为了消除指令 Cache 访问命中指令对指令访问性能改进有效性的稀释影响,我们测量了程序在 NIPS 及 BBIPS 两种场景下的指令访问缺失数,通过二者的比较直观的说明 BBIP 机制的预取效果.

Cache 缺失数(CMC)的相对偏差定义为

$$RB(CMC)=\frac{\sum CMC_{diff\_b}}{\sum_{b\in B}CMC_b} \tag{10}$$

其中,CMC_{diff_b}表示在基本块 b,由于使用 BBIP 预取减少的 Cache 访问缺失数;CMC_b则是在 NIPS 场景中,对基本块 b 的访问的总缺失数.如表 8 所示:随着 Cache 行大小的减小,所有程序的 CMC 相对偏差均急剧上升.当 Cache 行大小为 8 字节时,BBIP 预取大约可避免大约 70%~80%的指令 Cache 缺失.对比表 7 与表 8 的评估结果不难发现,CMC 反映的 Cache 性能改进比 SC 更显著.这是因为 *CMC* 相对偏差摒弃了 Cache 访问命中的影响;而在 SC 的相对偏差的计算过程中,Cache 访问命中的存在,稀释了 Cache 访问缺失减少导致的模拟周期的下降.表 8 的评估结果表明:对于典型的 Cache 行大小(如 32 字节),BBIP 预取技术能够有效减少指令 Cache 访问缺失.

Table 8 Relative bias of cache miss count (CMC) for different cache block size (the cache size is 4K bytes)

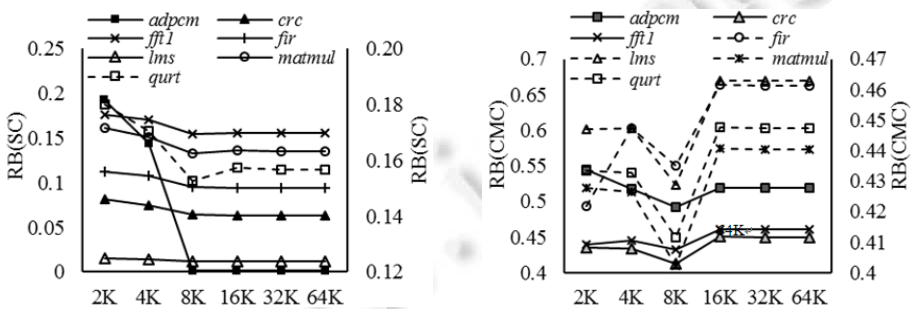
表 8 不同 Cache 行大小对应 Cache 缺失数(CMC)相对偏差(Cache 容量固定为 4K 字节)

RB(*CMC*)	256B	128B	64B	32B	16B	8B
adpcm	0.059	0.137	0.337	0.518	0.72	0.842
crc	0.08	0.154	0.264	0.434	0.634	0.79
*fft*1	0.069	0.09	0.232	0.446	0.633	0.791
fir	0.057	0.114	0.223	0.447	0.641	0.796
lms	0.083	0.134	0.257	0.447	0.648	0.801
matmul	0.071	0.113	0.244	0.426	0.63	0.788
qurt	0.032	0.107	0.242	0.433	0.636	0.791

如上所述,SC 的相对偏差是程序的指令 Cache 访问缺失率的函数.而 Cache 容量的大小直接影响程序的缺失率.接下来将讨论 Cache 容量与 SC 及 CMC 相对偏差间的关系,并以 32 字节的 Cache 行大小为例展开讨论.

如图 3(a)所示(其中,Cache 行大小为 32 字节,虚线标识的各测量值于副 *Y* 坐标轴(右侧)上表示):对于不同的 Cache 容量,BBIP 预取机制能够减少的模拟周期数对于程序 *lms* 几乎不变且极为有限,大约为 1.25%.这主要是由于 *lms* 具有较低的指令缺失率,指令 Cache 访问命中的时间在整个指令访问时间中占据主导地位,从而使得使用 BBIP 预取机制引起的模拟周期的变化对于整个指令访问时间的影响不显著.

测量结果还表明:当 Cache 容量增加到一定程度后,对于除 *adpcm* 以外的程序,SC 的相对偏差几乎保持不变.这是因为大部分测量程序具有较小的程序规模,当 Cache 容量达到一定值后继续增加并不能显著减少指令访问缺失次数.这种现象还表明,BBIP 对于不同容量的指令 Cache 均可提供较好的性能.对于 *adpcm*,当 Cache 容量从 2K 增加到 8K 字节时,模拟周期的相对偏差急剧下降.这是由于 *adpcm* 具有较大的程序规模,当指令 Cache 从 2K 增加到 8K 时,程序的指令访问缺失率从 5.08%急剧下降为 0.01%.



(a) 模拟周期数 SC (b) Cache 缺失数 CMC

Fig.3 Relative bias of simulation cycles (SC) and cache miss counts (CMC) for different cache size

图 3 不同 Cache 容量对应的模拟周期数 SC 及 Cache 缺失数 CMC 相对偏差

与模拟周期相比,Cache 访问缺失数能够更为直观地反映 BBIP 预取机制对于指令 Cache 访问性能的优化效果.如图 3(b)所示,BBIP 平均能够避免约 45.5%的指令访问缺失数.当 Cache 容量为 8K 字节时,CMC 相对偏差达到最小值.这是因为当 Cache 容量从 4K 增加到 8K 字节时,被测试程序的缺失率显著下降,从而导致指令缺失次数显著下降;同时,使用 BBIP 预取机制避免的缺失数却不太显著.而当 Cache 容量从 8K 字节继续增大至 16K 字节时,缺失率几乎保持不变且缺失次数较小,因此,BBIP 预取导致的缺失次数的减少将导致 CMC 相对偏差较

为显著的增加.当 Cache 容量从 32K 增加到 64K 字节时,指令 Cache 容量已大于程序规模,此时,冷启动缺失占据主导地位,冲突及容量缺失几乎不存在.此时,BBIP 预取机制能够避免的缺失数几乎保持不变,相对偏差也几乎恒定.

5.2 最坏执行情况下的Cache访问性能

对于实时系统而言,对其中运行的任务进行精确的 WCET 分析至关重要.BBIP 预取机制通过简化基本块的命中情况判定,提升了 WCET 分析中判定为命中的指令 Cache 访问数,从而减小程序的 WCET 评估值.接下来将讨论 BBIP 预取方法在减小 WCET 评估值方面的有效性.

评估过程使用 3 种不同的处理器配置,见表 9.其中,理想型分支预测器模型假设在程序执行过程中,所有分支均正确预测,此时,流水线中不存在由于分支预测错误引入的指令序列;2-level 分支预测器的分支历史表包含 1 024 个表项.

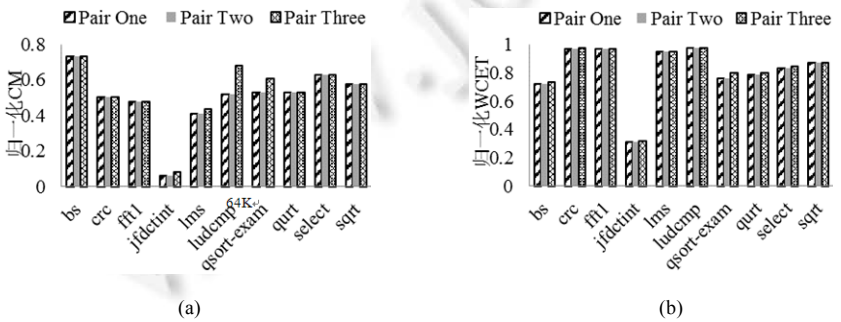
Table 9 Three processor configurations used in the evaluation

表 9 评估所用 3 组处理器配置

编号	流水线配置	分支预测器配置	指令 Cache 配置
1	按序执行	理想型	直接相联、32 字节行大小、1 024 字节容量
2	乱序执行	理想型	直接相联、32 字节行大小、1 024 字节容量
3	乱序执行	2-level	直接相联、32 字节行大小、1 024 字节容量

本文使用一些典型的实时基准测试程序对 BBIP 预取方法进行了有效性评估.这些测试程序来自 SNU 实时基准测试集[19],该测试集包含了实时应用的一些常用 Kernel 函数,能够较好地代表真实的实时应用的程序特性,因此为本领域国内外学者广泛使用.在评估过程中,测试程序通过定制的 GCC 编译器编译成 SimpleScalar 支持的 PISA(一种类 MIPS 指令集)指令.每条 PISA 指令长度为 8 字节.

在评估过程中,分别使用 NIPS 场景中获得的 WCET 评估值及指令缺失数为基准值对 BBIPS 中的对应值进行归一化处理.较小的归一化值意味着 BBIP 带来的相应指标的下降更为显著.如图 4(a)及图 4(b)所示(图例 Pair One(Two,Three)分别代表处理器配置 1(2,3)):对于 3 组处理器配置,使用 BBIP 预取机制后,所有程序的静态缺失次数与 WCET 评估值均有一定程度的改进(下降);且对于不同的测试程序,两者的改进程度不同.例如:使用 BBIP 预取机制后,程序 *jfdctint* 的静态缺失次数减少至 5.9%,而 *bs* 则为 73.3%.这主要是因为程序 *jfdctint* 的指令缺失率较高(约为 10.4%),指令访问缺失数较大,因此,使用 BBIP 预取机制后能够获得预取收益的基本块较多.与此对比,程序 *bs* 的指令访问数少(约为 64),缺失次数少(约为 9),因此,使用 BBIP 预取机制后缺失次数的减少有限.对于 3 种处理器配置,在使用 BBIP 预取机制后,测试程序的静态指令访问缺失数分别平均减小为原来的 49.7%,49.7%及 52.6%.



(a) (b)

Fig.4 Normalized instruction accesses determined statically as miss and WCET estimates in WCET evaluation in BBIPS

图 4 BBIPS 场景中使用 WCET 分析方法静态界定为缺失的归一化指令访问数及 WCET 评估值

通过比较图 4(a)与图 4(b)的评估结果可见:虽然静态缺失次数的减少对于 WCET 评估值具有直接影响,但是二者间并无成比例变化关系.例如:使用 BBIP 预取机制后,*bs* 的静态指令缺失数的减少(小于 30%)不如 *crc*(约为 50%)显著;但是 *bs* 对应的 WCET 评估值的改进较 *crc* 更显著,分别约为 30%及小于 2.5%.这是因为对于不同的程序,指令访问缺失数与缺失率不同,因此,指令访问性能对于程序运行时间的影响程度不同.当程序具有较高的缺失次数及缺失率时,指令缺失次数变化对于程序执行时间的影响显著.此时,指令缺失次数显著下降将导致 WCET 评估值的显著下降(如程序 *jfdctint*).而对于像 *crc* 这类程序,指令访问数大(约为 21 464)而缺失率小(约为 0.17%),此时,指令缺失次数的减少不能显著影响程序的运行时间,因此 WCET 评估值的改进有限.

从图 4(a)及图 4(b)还可得出,在 3 种不同的处理器配置下,BBIP 预取机制均能够有效降低指令访问缺失数及 WCET 评估值.但是对于一些程序,如 *qsort-exam*,使用乱序执行流水线及 2-level 分支预测器时,指令访问缺失数及 WCET 评估值的改进相对另外两种处理器配置较小.这主要是因为使用 BBIP 预取机制后,错误分支预测导致的指令 Cache 内容污染可能由于基本块后续存储块的提前加载而加剧.对于 3 种不同的处理器配置,在使用 BBIP 预取机制后,测试程序的 WCET 评估值分别平均减小为原来的 81.4%,81.4%及 82.35%.

在通用计算机系统中,Next-N-Line 指令预取技术因实现简单、硬件开销小、指令访问性能提升明显,受到广泛的关注.但是在实时系统中,Next-N-Line 指令预取技术中存在的无效预取造成的 Cache 内容污染,可能影响任务的 WCET 评估值.

接下来比较使用 BBIP 与 Next-N-Line 指令预取技术后程序对应的 WCET 评估值.评估过程使用直接映射、容量分别为 512 及 1 024 字节、行大小为 32 字节的指令 Cache 配置.在 Next-N-Line 指令预取机制的评估过程中,设定预取步长为 2 个 Cache 行,即,每次访问指令 Cache 时预取下两个 Cache 行.如图 5 所示:在绝大部分情况下,BBIP 指令预取技术能够提供比 Next-N-Line 预取技术更小的 WCET 评估值(其中,基准值为使用 Next-N-Line 预取时任务对应的 WCET 评估值,预取步长为 2 个 Cache 行).这是因为程序中存在跳转及循环指令,使用 Next-N-Line 预取的预取行可能不被执行;并且将当前位于指令 Cache 中的有用 Cache 行替换出去,而在 BBIP 指令预取机制中不存在这种无效预取.从图 5 还可看出:当使用 512 字节大小的指令 Cache 时,对于测试程序 *qsort-exam*,使用 Next-N-Line 预取能够提供比 BBIP 预取略小的 WCET 评估值.这是因为在 *qsort-exam* 中,顺序的指令访问较多,因此 Next-N-Line 预取中的无效预取较少;而相反地,BBIP 指令预取技术不能避免基本块第 1 条指令的访问缺失,因此,BBIP 指令预取技术对应的 WCET 评估值略大.平均而言:对于 512 字节及 1 024 字节指令 Cache,相对于 Next-N-Line 预取技术,使用 BBIP 预取技术能够使得 WCET 评估值分别降低约 9.2%及 8.8%.

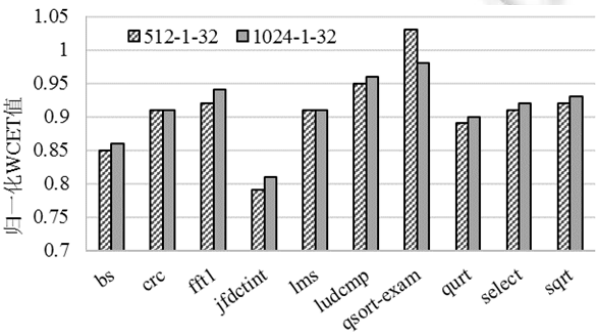


Fig.5 Normalized WCET estimates of BBIP prefetching with Next-N-Line prefetching as baseline

图 5 BBIP 预取相对于 Next-N-Line 指令预取的归一化 WCET 评估值

需要强调的是:在 Next-N-Line 预取方法 WCET 评估过程中,我们假设每次对指令 Cache 发起指令访问时均执行预取.基于该项假设,使用 Next-N-Line 预取后,Cache 访问缺失仅可能存在于某些基本块的第 1 条指令访问处,其甚至可以避免一些长度为 1 个存储块的基本块的指令访问缺失,而这在 BBIP 中无法保证.在真实硬件系统,这种访问时预取的实现方式通常会带来较大的访问时间及能耗开销,很少使用.因此,如果使用缺失时预取

的实现方案,则 BBIP 相对于 Next-N-Line 指令预取的 WCET 评估性能优势更为明显.另外,本文使用的测试程序规模较小,Next-N-Line 预取导致的 Cache 污染问题没有凸显.可以预见:随着实时任务程序规模的增加,Cache 污染问题的突出,BBIP 的优势将会更为明显.

6 结 论

为了简化 WCET 分析过程中指令 Cache 访问时间评估,减小实时任务 WCET 评估值,本文提出了一种基于程序基本块的指令预取方法(BBIP).在该预取方法中,每次指令访问发生缺失时,根据系统记录的基本块信息(如基本块起始地址及长度)对指令所在基本块的后续存储块执行指令预取.与现有应用于实时系统的指令预取技术相比,基于基本块的指令预取方法硬件实现开销小,且可有效避免无效预取及由此产生的指令 Cache 污染.另外,该预取方法能够极大地简化实时任务 WCET 分析,降低 WCET 评估值,从而改进实时系统任务可调度性分析.此外,与一些面向实时环境的新型 Cache 硬件结构相比,BBIP 预取方法能够以可选附属特性的形式存在于现有指令 Cache 结构中,适用性良好.

实时基准测试程序评估结果表明:BBIP 预取方法能够有效降低平均执行情况下的指令访问缺失数及任务执行时间;同时,在 WCET 分析中,具备 BBIP 预取功能的指令 Cache 能够将一部分原先静态判定为缺失的指令转化为访问命中,从而减少了静态指令访问缺失数,降低 WCET 评估值,有利于实时系统的可调度性分析.

致谢 在此,感谢新加坡国立大学发布了开源静态 WCET 评估工具 Chronos,并对其进行持续的支持与维护.此外,我们向对本文的工作给予支持和建议的同行,尤其是北京航空航天大学计算机学院的龙翔教授及其他老师和同学表示感谢.

References:

[1] Clifton C, Leavens GT, Chambers C, Millstein T. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In: Rosson MB, Lea D, eds. Proc. of the 2000 ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages & Applications. SIGPLAN Notices, 2000,35(10):130−145. [doi: 10.1145/353171.353181]

[2] Smith AJ. Sequential program prefetching in memory hierarchies. Computer, 1978,11(12):7−21. [doi: 10.1109/C-M.1978.218016]

[3] Smith AJ. Cache memories. ACM Computing Surveys (CSUR), 1982,14(3):473−530. [doi: 10.1145/356887.356892]

[4] Smith JE, Hsu WC. Prefetching in supercomputer instruction caches. In: Werner R, ed. Proc. of the 1992 ACM/IEEE Conf. on Supercomputing. IEEE Computer Society Press, 1992. 588−597. [doi: 10.1109/SUPERC.1992.236645]

[5] Pierce J, Mudge T. Wrong-Path instruction prefetching. In: Beaty S, Melvin S, eds. Proc. of the 29th Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO-29). IEEE, 1996. 165−175. [doi: 10.1109/MICRO.1996.566459]

[6] Joseph D, Grunwald D. Prefetching using Markov predictors. In: Pleszkun AR, Mudge T, eds. Proc. of the 24th Annual Int'l Symp. on Computer architecture. ACM Press, 1997,25(2):252−263. [doi: 10.1145/264107.264207]

[7] Luk CK, Mowry TC. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In: Bondi J, Smith J, eds. Proc. of the 31st Annual ACM/IEEE Int'l Symp. on Microarchitecture (MICRO-31). IEEE Computer Society Press, 1998. 182−194. [doi: 10.1109/MICRO.1998.742780]

[8] Xia C, Torrellas J. Instruction prefetching of systems codes with layout optimized for reduced cache misses. In: Baer JL, ed. Proc. of the 23rd Annual Int'l Symp. on Computer Architecture. 1996,24(2):271−282. [doi: 10.1109/ISCA.1996.10019]

[9] Reinman G, Calder B, Austin T. Fetch directed instruction prefetching. In: Ronen R, Farrens M, Spillinger I, eds. Proc. of the 32nd Annual Int'l Symp. on Microarchitecture (MICRO-32). IEEE, 1999. 16−27. [doi: 10.1109/MICRO.1999.809439]

[10] Zhang Y, Haga S, Barua R. Execution history guided instruction prefetching. In: Ebcioglu K, Pingali K, Nicolau A, eds. Proc. of the 16th Int'l Conf. on Supercomputing. ACM Press, 2002. 199−208. [doi: 10.1145/514191.514220]

[11] Aamodt TM, Chow P, Hammarlund P, Wang H, Shen JP. Hardware support for prescient instruction prefetch. In: Tirado F, Zapata EL, eds. Proc. of the IEEE Int'l Symp. on High Performance Computer Architecture. IEEE Computer Society, 2004. 84. [doi: 10.1109/HPCA. 2004.10028]

[12] Yan J, Zhang W. WCET analysis of instruction caches with prefetching. In: Pande S, Li ZY, eds. Proc. of the 2007 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems. ACM SIGPLAN Notices, 2007,42(7): 175−184. [doi: 10.1145/1254766.1254801]

[13] Ding Y, Yan J, Zhang W. Optimizing instruction prefetching to improve worst-case performance for real-time applications. Journal of Computing Science and Engineering, 2009,3(1):59−71. [doi: 10.5626/JCSE.2009.3.1.059]

[14] Lee M, Min SL, Kim CS. A worst case timing analysis technique for instruction prefetch buffers. Microprocessing and Microprogramming, 1994,40(10):681−684. [doi: 10.1016/0165-6074(94)90017-5]

[15] Schoeberl M. A time predictable instruction cache for a Java processor. In: Zahir T, Angelo C, eds. Proc. of the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops. Berlin, Heidelberg: Springer-Verlag, 2004. 371−382. [doi: 10.1007/978-3-540-30470-8_52]

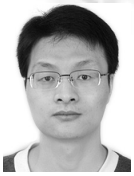
[16] Li XF, Roychoudhury A, Mitra T. Modeling out-of-order processors for WCET analysis. Journal of Real-time Systems, 2006,34(3): 195−227. [doi: 10.1007/s11241-006-9205-5]

[17] Austin T, Larson E, Ernst D. SimpleScalar: An infrastructure for computer system modeling. Computer, 2002,35(2):59−67. [doi: 10.1109/2.982917]

[18] Arnold R, Mueller F, Whalley D, Harmon M. Bounding worst-case instruction cache performance. In: Juan S, Rico P, eds. Proc. of the IEEE Real-Time Systems Symp. 1994. 172−181. [doi: 10.1109/REAL.1994.342718]

[19] Homepage of SNU real-time benchmark suite. 2007. http://archi.snu.ac.kr/realtime/benchmark/

王恩东(1964－),男,山东济南人,教授,CCF高级会员,主要研究领域为计算机系统结构,混合存储系统,可扩展系统互连协议分布对象计算.



倪璠(1984－),男,博士,工程师,主要研究领域为计算机系统结构,实时系统,操作系统.

陈继承(1976－),男,博士,高级工程师,CCF会员,主要研究领域为计算机体系结构,缓存一致性,集成电路设计.

王洪伟(1982－),男,博士,工程师,主要研究领域为计算机体系结构,高性能计算,人工智能.

唐士斌(1986－),男,博士,工程师,主要研究领域为计算机体系结构,缓存一致性,片上存储系统.